

Architecture-Adaptive Routability-Driven Placement for FPGAs

Akshay Sharma¹, Carl Ebeling², Scott Hauck¹

¹ Dept. of Electrical Engineering, ²Dept. of Computer Science & Engineering
University of Washington, Seattle, WA – 98195, USA
akshay@ee.washington.edu, ebeling@cs.washington.edu, hauck@ee.washington.edu

Abstract. Current FPGA placement algorithms estimate the routability of a placement using *architecture-specific* metrics. The shortcoming of using architecture-specific routability estimates is limited adaptability. A placement algorithm that is targeted to a class of architecturally similar FPGAs may not be easily adapted to other architectures. The subject of this paper is the development of a routability-driven architecture adaptive FPGA placement algorithm called Independence. The core of the Independence algorithm is a simultaneous place-and-route approach that tightly couples a simulated annealing placement algorithm with an architecture adaptive FPGA router (Pathfinder). The results of our experiments demonstrate Independence’s adaptability to island-style FPGAs, a hierarchical FPGA architecture (HSRA), and a coarse-grained reconfigurable architecture (RaPiD). The quality of the placements produced by Independence is within 1.2% of the quality of VPR’s placements, 17% better than the placements produced by HSRA’s placer, and within 0.7% of RaPiD’s placer. Further, our results show that Independence produces clearly superior placements on routing-poor island-style FPGA architectures.

1 Introduction

The most important architectural feature of an FPGA is arguably its interconnect structure. Since any FPGA has a finite number of discrete routing resources, a large share of architectural research effort is devoted to determining the composition of an FPGA’s interconnect structure. During architecture development, the effectiveness of an FPGA’s interconnect structure is evaluated using placement and routing tools (collectively termed place-and-route tool). The place-and route tool is responsible for producing a physical implementation of an application netlist on the FPGA’s prefabricated hardware. Specifically, the placer determines the actual physical location of each netlist logic block in the FPGA layout, and the router assigns the signals that connect the placed logic blocks to routing resources in the FPGA’s interconnect structure. Due to the finite nature of an FPGA’s interconnect structure, the success of the router is heavily reliant on the quality of the solutions produced by the placer. Not surprisingly, the primary objective of the placer is to produce a placement that can indeed be routed by the router.

The effectiveness of a placement tool as an evaluation mechanism relies on the ability of the placement algorithm to capture the FPGA’s interconnect structure. Cur-

rently, the *modus operandi* used in the development of placement algorithms is to use *architecture-specific* metrics to heuristically estimate the routability of a placement. For example, the routability of a placement on island-style FPGAs is estimated using the ever-popular Manhattan Distance wirelength metric, while the routability of a placement on tree-based architectures is estimated using cutsizes metrics.

Architecture-specific routability estimates limit the adaptability of a placement algorithm. To the best of our knowledge, there is no single placement approach that can adapt effectively to the interconnect structure of every FPGA in the architecture spectrum. This often proves to be an impediment in the early stages of FPGA architecture development, when the targeted placement algorithm is not well defined due to a lack of architectural information. We feel that research in FPGA architectures would stand to benefit from a universal placement algorithm that can quickly be retargeted to relatively diverse FPGA architectures.

The subject of this paper is the development of an architecture-adaptive routability-driven FPGA placement algorithm called Independence. The algorithm's adaptability is a direct result of using the Pathfinder [10] algorithm to calculate the cost of a placement. Specifically, we use Pathfinder in the inner loop of a simulated annealing placement algorithm to maintain a fully routed solution at all times. Thus, instead of using architecture-specific routability estimates, we use the routing produced by an architecture adaptive router to guide the algorithm to a routable placement.

The remainder of this paper is organized as follows. Section 2 examines VPR's routability-driven placement cost function, and uses architectural examples to demonstrate limitations of VPR's cost formulation. Section 3 describes Independence's placement heuristic, and important aspects of integrating Pathfinder with Independence. We present our validation strategy and experimental results in Section 4. Section 5 briefly discusses Independence's runtime characteristics, and Section 6 concludes the paper.

2 VPR Targets Island-style FPGAs

VPR [2,3] is the current, public-domain state-of-the-art FPGA placer. VPR's core is a simulated annealing [8] placement algorithm that uses a net semi-perimeter metric to estimate the routability of a placement. VPR consistently produces high-quality placements, and at the time of this writing, the best reported placements for the Toronto20 [1] benchmark netlists are those produced by VPR.

Due to a strong prevalence of routing rich island-style FPGA architectures, VPR's placement algorithm is primarily targeted to island-style FPGAs. The semi-perimeter based cost function relies on certain defining features of island-style FPGAs (Fig. 1):

Two-dimensional Geometric Layout - An island-style FPGA is laid out as a regular two-dimensional grid of logic blocks surrounded by a sea of routing wires and switches. As a result, VPR's cost function is based on the assumption that the routability of a net is proportional to the Manhattan distance (measured by semi-perimeter) between its terminals. A net with terminals that are far apart needs more routing resources than a net with terminals close to each other. A direct result of a semi-

perimeter based cost function is tightly packed placements, even if the capacity of the target FPGA far exceeds the logic requirements of the netlist.

Uniform Connectivity – Island-style architectures provide uniform connectivity. The number and type of routing resources available for a net with a given semi-perimeter are largely *independent* of the actual placement of the terminals of the net. Thus, VPR determines the cost of a net based purely on its semi-perimeter, and not the actual location of the terminals of the net.

Directionality – Island-style architectures have no implied directionality. The routing structure does not impose constraints on the placement of logic blocks. Thus, no move made by VPR's simulated annealing algorithm is illegal. As long as a placement is valid (no overlapping logic blocks), an island-style architecture guarantees that a route exists between any two logic blocks regardless of their locations.

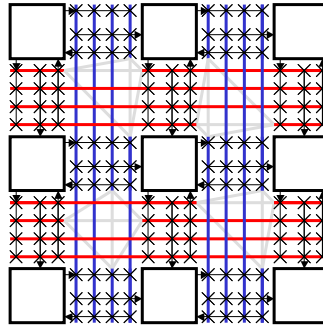


Fig. 1. An illustration of an island-style FPGA. The white boxes represent logic blocks. The horizontal and vertical intersecting boldface lines represent routing wires. The logic blocks connect to surrounding wires using programmable connection-points (shown as crosses), and individual wires connect to each other by means of programmable routing switches (shown as gray lines)

VPR's dependence on island style FPGA architectures limits its adaptability to architectures that do not provide all the features of island-style FPGAs (Fig. 2). For instance, the interconnect structure of an FPGA architecture may not conform to the Manhattan distance estimate of routability. One example is the hierarchical interconnect structure found in tree-based FPGA architectures [6]. Another class of non-island style FPGA architectures provides heterogeneous interconnect structures. Triptych [4] is an example that provides different types of routing resources in the horizontal and vertical directions, while the architecture in [7] provides horizontal routing channels that gradually increase in width from left to right. Finally, efforts to incorporate FPGA-like logic in System-on-Chip designs have motivated non-rectangular FPGA fabrics. The FPGA fabrics proposed in [14] are built by abutting smaller, rectangular fabrics of different aspect ratios.

The architectural examples cited in this section clearly show that a semi-perimeter placement cost function does not adapt well to non-island style FPGAs. A cost function's adaptability lies in its ability to guide a placement algorithm to a high-quality solution *across* a range of architecturally diverse FPGAs. In the next section we de-

scribe Independence, an architecture adaptive routability-driven FPGA placement algorithm.

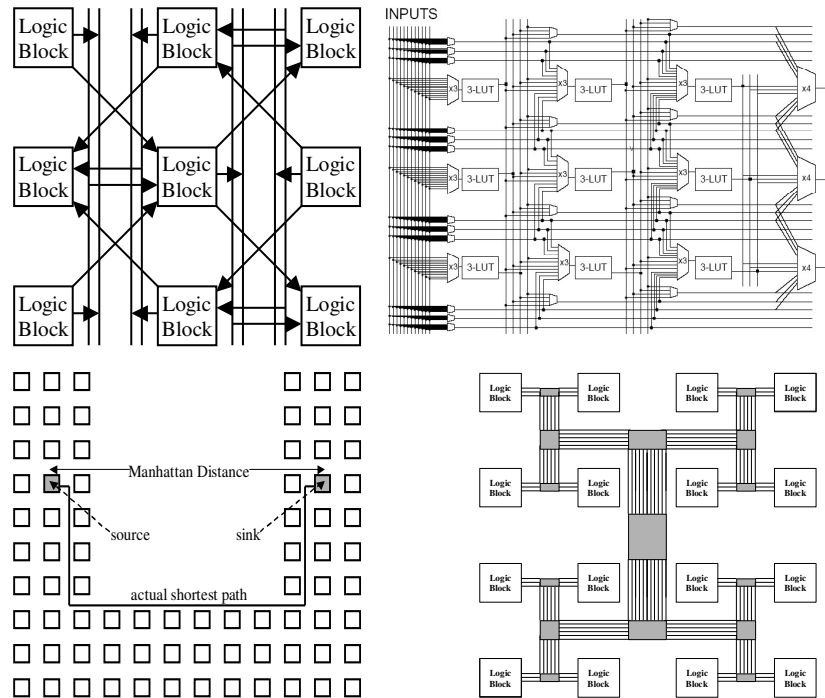


Fig. 2. Non Island-style FPGAs. Clockwise from top left: Triptych [4], directional architecture from [7], a hierarchical FPGA [6], and a U-shaped FPGA core [14].

3 Independence

The core of the Independence algorithm is tightly integrated placement and routing. Instead of using architecture specific heuristics, we estimate routability during placement by actually routing signals using an *adaptive* routing algorithm (Pathfinder). By doing so, we expect accurate estimates of the routing resource usage and total congestion of a placement while maintaining the adaptability of the placement algorithm.

Placement Heuristic and Cost Formulation – Since simulated annealing has clearly produced some of the best placement results reported for FPGAs [3], we chose to use simulated annealing as Independence’s placement heuristic. Independence’s cooling schedule is mostly an adoption of VPR’s cooling schedule. This is because VPR’s cooling schedule is adaptive, and incorporates some of the most powerful features from earlier research in cooling schedules. For similar reasons, we chose an

auto-normalizing formulation for Independence’s cost function. The main benefit of using normalization variables is that changes in cost of a placement do not depend on the actual *magnitude* of the cost variables. This makes the cost function adaptive, since the size of a netlist or the target architecture does not skew cost calculations. Independence’s cost function is described in equation (3).

$$\Delta C = \Delta \text{WireCost} / \text{prevWireCost} + \lambda * \Delta \text{CongestionCost} / \text{CongestionNorm} \quad (3)$$

WireCost: The wire cost of a placement (equation (4)) is calculated by summing the number of routing resources used by each signal in the placed netlist. In equation (4), N is the number of signals in the netlist, and $\text{NumRoutingResources}_i$ is the number of routing resources in the route tree of signal i . The normalization variable *prevWireCost* in equation (3) is equated to the *WireCost* of a placement before a placement move is attempted.

$$\text{WireCost} = \sum_{i=1}^N \text{NumRoutingResources}_i \quad (4)$$

CongestionCost: The congestion cost (equation (5)) represents the extent to which the routing resources are congested in a given placement, and is calculated by summing the number of signals that overuse each congested resource. In equation (5), Occupancy_i is the number of signals that are currently using routing resource i , Capacity_i is the capacity of routing resource i , and R is the total number of vertices in the routing graph of the target architecture. It could be argued that *CongestionCost* renders *WireCost* redundant, since the objective of an FPGA placement algorithm is to produce a routable netlist. However, a cost function that is unaware of changes in wire cost will not recognize moves that might improve future congestion due to reductions in routing resource usage. Also, note that the total congestion cost of the placement cannot be used as a normalizing factor, since *CongestionCost* might be zero towards the end of the annealing process. In our present implementation, *CongestionNorm* (equation (3)) is equated to *prevWireCost*.

$$\text{CongestionCost} = \sum_{i=1}^R \max(\text{Occupancy}_i - \text{Capacity}_i, 0) \quad (5)$$

λ : This tuning parameter controls the relative importance of changes in wire and congestion costs, and is a number greater than one. The magnitude of λ is inversely related to the richness of the target architecture’s interconnect structure.

Integrating Pathfinder – FPGA routing is a computationally intensive process. Admittedly, it is infeasible to reroute *all* the signals in a netlist after each placement move. Our solution is to start out with an initially complete routing, and then incrementally reroute signals during placement. Specifically, only the signals that connect to the logic blocks involved in a move are ripped up and rerouted (Fig. 3). This is based on the intuition that for any given move, major changes in congestion and rout-

ing resource usage will be primarily due to the rerouting of signals that connect moved logic blocks.

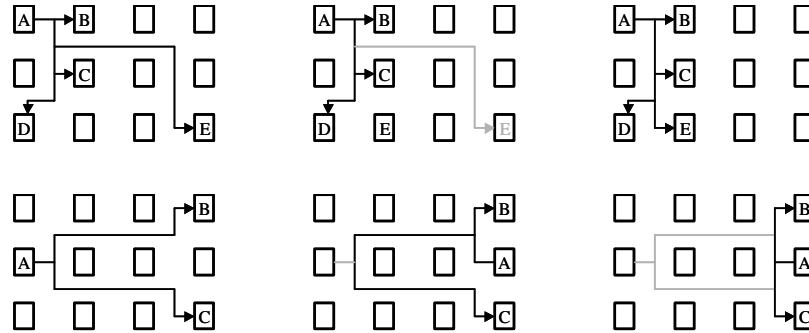


Fig. 3. Top – Sink E is moved immediately to the right of sink D. In this case, only E’s input branch (shown in gray) is ripped up and rerouted. Bottom – The source terminal A is moved to the location between sinks B and C. If we reroute from A to the partial route-tree, the resultant routing would require far more routing than necessary. Ripping up and rerouting the entire net produces a better routing.

Since we only attempt an incremental rip-up and reroute after every move, the routes found for signals during the early parts of an annealing iteration may not accurately reflect the congestion profile of the placement at the end of an iteration. Hence, we periodically refresh the netlist’s routing by ripping up and rerouting all signals. Currently, the netlist is ripped up and rerouted at the end of every temperature iteration.

In light of the fact that the placement of a netlist is constantly changing during simulated annealing, it is necessary to examine whether Pathfinder’s cost function is directly applicable to finding routes during incremental rip-up and reroute. When routing a signal, Pathfinder uses the number of signals currently sharing a routing node (*presentSharing*), and the history of congestion on the node (*historyCost*) to calculate the cost of the routing node. Since the netlist is completely routed at any given point in the placement process, the current sharing of routing nodes can easily be calculated, and thus we directly adopt Pathfinder’s *presentSharing* cost term.

Pathfinder’s history cost term is motivated by the intuition that routing nodes that have been historically congested during the routing process probably represent a congested area of the placed netlist. Thus, if a routing node is shared at the end of a routing iteration, its history cost is incremented by a fixed amount to make the node more expensive during subsequent iterations. Note that the process of updating history costs during a Pathfinder run makes history cost a monotonically increasing function. A monotonically increasing history cost formulation is inappropriate for Independence. An increasing history cost would reflect the congestion on a routing node during the *entire* placement process. However, since placements are in constant flux during the placement process, the congestion on a routing node during the early stages of the annealing process (when placements are very different) might not be relevant to the routing process towards the end.

Independence uses a decaying function to calculate history costs during incremental rip-up and reroute. Specifically, we use a mathematical formulation that decreases the relevance of history information from earlier parts of the placement process. Currently, we update history costs once every temperature iteration based on the assumption that the number of signals ripped up and rerouted during a temperature iteration is roughly equivalent to the number of signals routed during a single or small number of Pathfinder iterations. The history cost of a routing node during a temperature iteration ‘ $i+1$ ’ is presented in equation (6).

$$\begin{aligned}
 & \text{if (shared)} \\
 & \quad \text{historyCost}_{i+1} = \alpha * \text{historyCost}_i + \beta \\
 & \text{else} \\
 & \quad \text{historyCost}_{i+1} = \alpha * \text{historyCost}_i
 \end{aligned} \tag{6}$$

In equation (6), i is a positive integer, and α and β are empirical parameters. Currently, $\alpha = 0.9$ and $\beta = 0.5$. Thus, the history cost of a shared routing node during a new iteration is determined by 90% of the history cost during earlier iterations plus a small constant. As an example, the history cost of a node that is shared during the first five iterations progressively goes from 0 to 0.5, to 0.95, to 1.36, and to 1.72. In cases where a routing node is not shared during a temperature iteration, its history cost is allowed to decay as per equation (6).

4 Results

The objective of our experiments was to demonstrate Independence’s adaptability to FPGAs that have clearly different interconnect architectures. We selected three architectures to validate our claim that Independence is in fact an architecture-adaptive placement algorithm. We present our results for each of these three architectures in this section.

Hierarchical Interconnect Structures (HSRA)

Our first experiment (**Experiment 1**) targets HSRA [6], which has a hierarchical, tree-based interconnect structure (Fig. 4). The richness of HSRA’s interconnect structure is defined by its *base channel width* and *interconnect growth rate*. The base channel width ‘ c ’ is the number of tracks at the leaves of the interconnect tree (in Fig. 4, $c=3$). The growth rate ‘ p ’ is the rate at which the interconnect grows towards the root (in Fig. 4, $p=0.5$). The growth rate is realized using the following types of switch-blocks:

Non-compressing (2:1) switch blocks – The number of root-going tracks is equal to the sum of the number of root-going tracks of the two children.

Compressing (1:1) switch blocks – The number of root-going tracks is equal to the number of root-going tracks of either child.

A repeating combination of non-compressing and compressing switch blocks can be used to realize any value of p less than one. So, a repeating pattern of (2:1 \rightarrow 1:1) switch blocks realizes $p=0.5$, while the pattern (2:1 \rightarrow 2:1 \rightarrow 1:1) realizes $p=0.67$. In

HSRA, each logic block has a single LUT/FF pair. The input-pin connectivity is based on a *c-choose-k* strategy [6], and the output pins are fully connected. The base channel width of the target architecture is eight, and the interconnect growth-rate is 0.5. The base channel width and interconnect growth rate were both selected so that the placements produced by HSRA’s CAD tool were noticeably depopulated (a medium-stress placement problem). Fig 4 (right) compares the minimum base channel widths required to route placements produced by HSRA’s placer and Independence. Overall, Independence produced placements that were 17% better than HSRA’s placer.

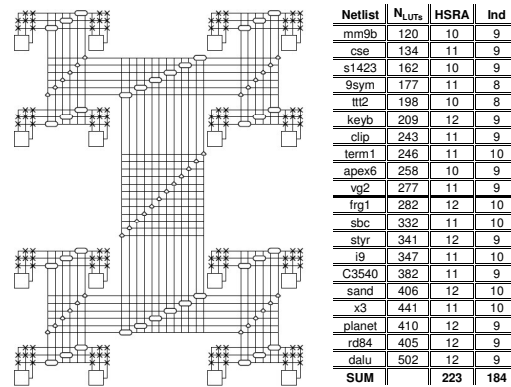


Fig. 4. An illustration of HSRA’s interconnect structure is on the left, and the minimum base-channel widths required to route placements produced by HSRA’s placer and Independence are on the right.

The RaPiD Architecture

Our second experiment (**Experiment 2**) targeted the RaPiD [5] architecture (Fig. 5). RaPiD is a coarse-grained 1-dimensional architecture targeted to streaming, compute-intensive applications. The logic structure contains 16-bit registers, ALUs, multipliers and small SRAM blocks. RaPiD’s interconnect structure consists of segmented 16-bit buses. There are two types of buses; *short* buses provide local communication between logic blocks, while *long* buses can be used to establish longer connections using bidirectional switches called *bus-connectors* (shown as the small square boxes in Fig. 5). RaPiD’s interconnect structure is relatively constrained because there is no inter-bus switching capability in the interconnect structure. A bus-connector can only be used to connect the two bus-segments incident to it. Thus, RaPiD is an interesting candidate architecture for a routability-driven placement algorithm.

Island-style FPGAs

Our third experiment (**Experiment 3**) compared the placements produced by Independence with VPR when targeted to a clustered, island-style architecture. Each logic block cluster in this architecture has eighteen inputs, eight outputs, and eight 4-LUT/FF pairs per cluster. The interconnect structure consists of staggered length four track segments and disjoint switchboxes. The input pin connectivity of a logic block

cluster is $0.4*W$ (where W is the channel width) and output pin connectivity is $0.125*W$. The island-style architecture described here is similar to the optimal architecture reported in [9]. The table on the left in Fig. 6 shows that the placements produced by VPR and Independence were within 1.2% of each other. Note that each netlist is placed on the *minimum size* square array required to just fit the logic and/or IO blocks in a netlist.

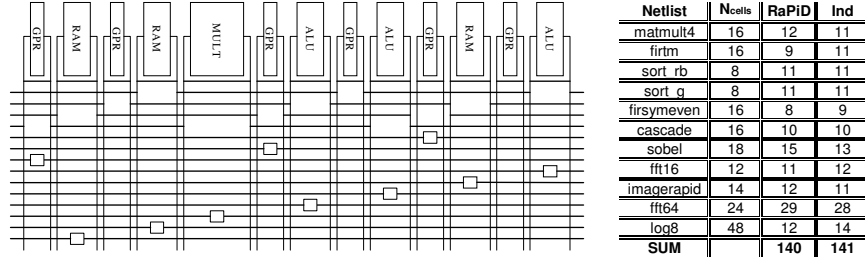


Fig. 5. A RaPiD cell (left). In the table on the right, the N_{cells} column lists the min number of RaPiD cells required to fit the netlist. The RaPiD column lists the min track-count required by placements produced by the placer described in [12], while the Ind column lists the min track-count required to route placements produced by Independence. Overall, the min track-counts required by RaPiD’s placer and Independence were within 0.7%.

Our final experiment (**Experiment 4**) studied Independence’s adaptability to routing-poor FPGA architectures. The philosophy behind routing-poor architectures [4,6] is increased silicon utilization through efficient use of the interconnect structure (which often accounts for ~90% of the total area in current FPGA families). Routing-poor architectures attempt to increase interconnect utilization at the expense of logic utilization.

The table on the right in Fig. 6 shows the extent to which Independence is able to adapt to routing-poor island-style FPGAs. The parameters of the target array are identical to those used in **Experiment 3**. The only exception is the logic capacity, which is four times (the width and height of the target array are each 2X the minimum required to fit the netlist) that of a minimum size square array. Column 1 lists the netlists used in the experiment, and column 2 lists the minimum track counts needed by VPR to route each netlist. Let the minimum track count needed by VPR to route a netlist be W_{VPR} . Columns 3 through 8 list the number of tracks in a target architecture that has $1.0*W_{\text{VPR}}$, $0.9*W_{\text{VPR}}$, $0.8*W_{\text{VPR}}$, $0.7*W_{\text{VPR}}$, $0.6*W_{\text{VPR}}$, and $0.5*W_{\text{VPR}}$ tracks respectively. In Columns 3 – 8, a lightly shaded table entry (black text) means that Independence produced a routable placement on that architecture, while a dark shaded entry (white text) means that Independence was unable to produce a routable placement. So, for example, the lightly shaded table entry 37 for the netlist *ex5p* means Independence produced a routable placement for *ex5p* on a 37-track ($0.7*52$) architecture. Similarly, the dark shaded entry 32 for *ex5p* means that Independence failed to produce a routable placement for *ex5p* on a 32-track ($0.6*52$) architecture. The results in Fig. 6 show that on routing-poor island-style FPGAs, Independence produced up to 40% better placements than VPR. Note that VPR does not possess the ability to adjust to routing-poor architectures, and thus cannot use the extra space to reduce track

count. Further, since the height and width of the target array in **Experiment 4** is approx 2X the minimum required, the target arrays in **Experiments 3** and **4** have the same bisection bandwidth. Thus, the point at which Independence fails (at ~50% VPR’s track count) probably represents a lower-bound on the minimum achievable track-count.

Netlist	N _{blocks}	VPR	Ind
s1423	51	17	17
term1	77	17	17
vda	122	33	33
dalu	154	25	25
x1	181	22	23
apex4	193	60	60
i9	195	19	19
misex3	207	45	47
ex5p	210	60	61
alu4	215	39	40
x3	290	26	25
rot	299	27	28
tseng	307	34	36
pair	380	36	36
dsip	598	31	30
SUM		491	497

Netlist	N _{blocks}	VPR	1	0.9	0.8	0.7	0.6	0.5
s1423	51	17	17	16	14	12	11	9
vda	122	33	33	30	27	24	20	17
rot	299	30	30	27	24	21	18	15
alu4	215	37	37	34	30	26	23	19
misex3	207	43	43	39	35	31	26	22
ex5p	210	52	52	47	42	37	32	26
tseng	307	33	33	30	27	24	20	17
apex4	193	52	52	47	42	37	32	26
diffeq	292	31	31	28	25	22	19	16
dsip	598	34	34	31	28	24	21	17

Fig. 6. The results of **Experiment 3** are presented in the table on the left. The table on the right presents the results of **Experiment 4**.

6 Conclusions & Future Work

The results of the experiments presented in Section 4 demonstrate Independence’s adaptability to three significantly different interconnect styles. Further, our experiment with routing-poor island-style FPGAs showed that Independence is appropriately sensitive to the richness of interconnect structures. When considered together, the results presented in Section 4 are a clear validation of using an architecture-adaptive router to guide FPGA placement. We believe that a production version of Independence (i.e. a well-engineered version that has been enhanced to reduce runtime) would be of considerable use in the following scenarios:

Architecture Evaluation: Independence’s adaptability makes it a naturally attractive candidate for evaluating FPGA interconnect structures during the early stages of FPGA architecture development. Independence’s Pathfinder-based approach is particularly useful for this task because its history cost formulation naturally identifies congestion bottlenecks in the interconnect structure.

Evaluation of CAD Tools: In many cases, CAD tool developers spend considerable time trying to evaluate the “goodness” of an architecture-specific placement tool. The central concern in this process is finding an alternative comparison point *without* resorting to impractical exponential search strategies that attempt to find an optimal solution. The quality of the placements obtained on targeting Independence to the architecture would serve as a good quality goal during the tool development process.

Currently, the runtime penalty incurred by Independence’s simultaneous place-and-route technique is significant. For example, Independence is *four orders of magnitude* slower than VPR. Since a small number of routing searches are launched during every attempted placement move, Independence’s runtime is directly impacted by

the size of the routing graph *and* the size of the netlist. In contrast, the runtime of an architecture-specific placement algorithm like VPR depends only on the size of the netlist, and is not affected by the size of the target architecture.

The current version of Independence is merely a proof-of-concept implementation. This version was developed to primarily demonstrate the adaptability of a simulated annealing placement algorithm that uses the Pathfinder algorithm to calculate the cost of a placement. Independence's current incarnation may require multiple runtime enhancements before it can be considered a production version. In the near future, we will actively consider algorithmic (A* search), statistical and empirical techniques to speed up Independence.

Acknowledgment

We would like to thank Andre' DeHon at Caltech for providing the HSRA toolflow and helping us understand various aspects of the architecture. Thanks are also due to Larry McMurchie for his helpful comments and feedback during the development of the Independence algorithm. This work was supported by grants from NSF and Altera, Inc. Scott Hauck was supported in part by an NSF Career Award and an Alfred P Sloan Fellowship.

References

1. V. Betz, <http://www.eecg.toronto.edu/~vaughn/challenge/challenge.html>.
2. V. Betz and J. Rose, "VPR: A New Packing, Placement and Routing Tool for FPGA Research", 7th *International Workshop on Field-Programmable Logic and Applications*, pp 213-222, 1997.
3. V. Betz, J. Rose and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*, Kluwer Academic Publishers, Boston, MA:1999.
4. G. Boriello, C. Ebeling, S Hauck, S. Burns, "The Triptych FPGA Architecture", *IEEE Transactions on VLSI Systems*, Vol. 3, No. 4, pp. 473 – 482, 1995.
5. D. Cronquist, P Franklin, C Fisher, M Figueroa, and C. Ebeling, "Architecture Design of Reconfigurable Pipelined Datapaths", *Twentieth Anniversary Conference on Advanced Research in VLSI*, pp 23 – 40, 1999.
6. A. DeHon, "Balancing Interconnect and Computation in a Reconfigurable Computing Array (or, why you don't really want 100% LUT utilization)," *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp 69 – 78, 1999.
7. N. Kafafi, K. Bozman, S Wilton, "Architectures and Algorithms for Synthesizable Embedded Programmable Logic Cores", *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 3 – 11, 2003.
8. S. Kirkpatrick, C. Gelatt Jr., M. Vecchi, "Optimization by Simulated Annealing", *Science*, 220, pp. 671-680, 1983.
9. A. Marquardt, V. Betz and J. Rose, "Speed and Area Tradeoffs in Cluster-Based FPGA Architectures", *IEEE Transactions on VLSI Systems*, Vol. 8, No. 1, pp. 84 – 93, 2000.
10. L. McMurchie and C. Ebeling, "PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs", *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp 111-117, 1995.

11. C. Sechen, *VLSI Placement and Global Routing Using Simulated Annealing*, Kluwer Academic Publishers, Boston, MA: 1988.
12. A. Sharma, "Development of a Place and Route Tool for the RaPiD Architecture", *Master's Project, University of Washington*, December 2001.
13. J. Swartz, V. Betz, J. Rose, "A Fast Routability-Driven Router for FPGAs", *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 140 – 149, 1998.
14. T. Wong, "Non-Rectangular Embedded Programmable Logic Cores", *M.A.Sc. Thesis, University of British Columbia*, May 2002.